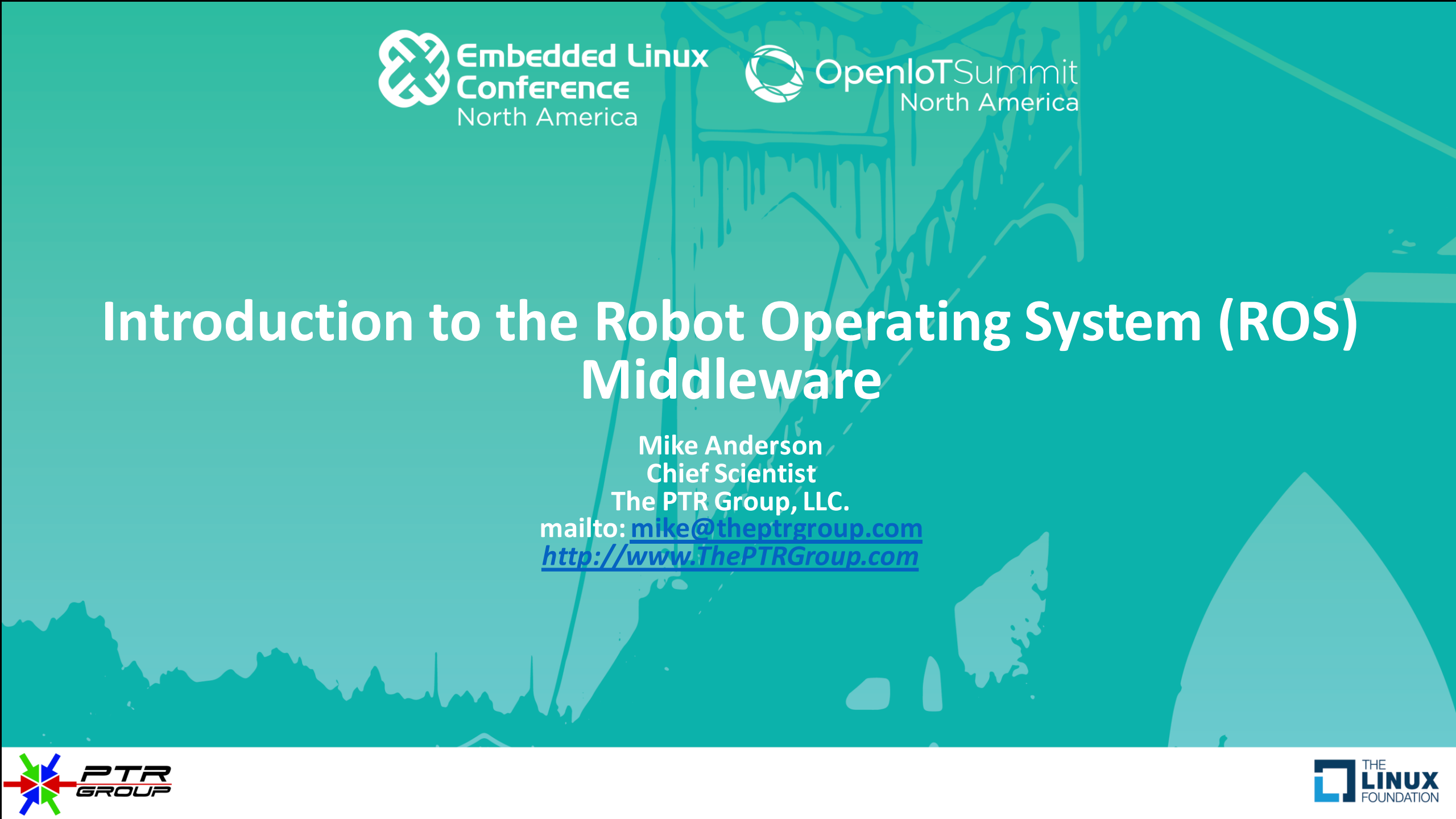


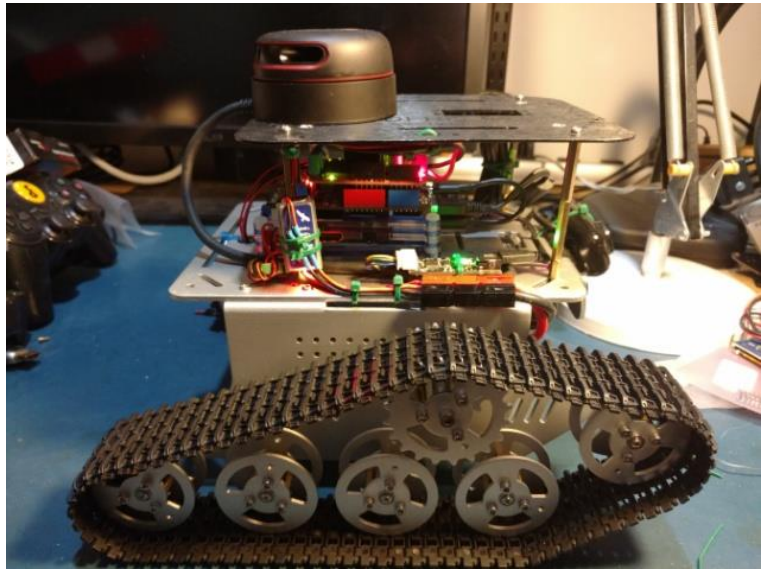
Introduction to the Robot Operating System (ROS) Middleware

Mike Anderson
Chief Scientist
The PTR Group, LLC.
mailto: mike@theptrgroup.com
http://www.ThePTRGroup.com

What We Will Talk About…

- What is ROS?
- Installing ROS
- Testing your installation
- ROS components
- ROS concepts
- Computation graph and naming conventions
- Your first robot
- Pub/Sub example
- Summary

What is ROS?



- The Robot Operating System is a collaborative effort to create a robust, general purpose mechanism for creating applications for robotics
  - Why? Because robotics control software is hard!
- Things that seem trivial to a human can be wildly hard for a robot
  - Just think about turning a door knob to open a door or walking up steps…
- There are so many different robotic applications, no one individual, company, university or laboratory could possibly enumerate all of the options
  - ROS is the culmination of the underlying infrastructure for robotic control, a robust set of tools, a collection of capabilities than can be mixed and matched and a broad community ecosystem of developers working on specific topic areas

History and Legacy

- Started in 2007 as an outgrowth of the STanford AI Robot (STAIR) and Personal Robots (PR) programs from Stanford University in Stanford, CA
- Sponsored by an local robotics incubator named Willow Garage
  - Willow Garage produced a robot known as the PR2
    - The purchasers of the PR2 became a loose federation of developers each contributing their code back to the greater community
- Licensed under the permissive BSD open-source license
  - However, some modules have licenses like ASLv2, GPLv2, MIT, etc.
- Latest release is “Lunar Loggerhead” in May of 2017
- ROS is supported by the Open Source Robotics Foundation
  - https://www.osrfoundation.org/

Source: willow garage.com

Installing ROS

- Native ROS installation of either Kinetic Kame or Lunar Loggerhead is supported out of the box for Debian-based distributions such as Ubuntu, Linux Mint, Debian and derivative distributions
  - Some experimental support for Gentoo, macOS and Yocto
- Pretty much your typical add GPG key, add apt sources, apt-get update, apt-get install sequence found with Debian PPAs etc.
  - http://wiki.ros.org/lunar/Installation/Ubuntu

Source: ros.org

Next Steps…

- After the initial installation, you will need to initialize **rosdep** and set your environment variables

```
$ sudo rosdep init
$ rosdep update
```

- Then take care of the environment:

```
$ echo "source /opt/ros/lunar/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

- In order to be able to build ROS packages, you'll need some additional dependences:

```
$ sudo apt-get install python-rosinstall
python-rosinstall-generator python-wstool build-essential
```

- Now, you're ready to test the installation

Testing the Installation with a simple build

- The ROS build system is called catkin
  - The name *catkin* comes from the tail-shaped flower cluster found on willow trees -- a reference to Willow Garage where catkin was created
- At this point, you're ready to try a simple build:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

- Even though the workspace is empty, you can still issue a make

```
$ cd ~/catkin_ws
$ catkin_make
```

Core ROS Components

- At its core, ROS is an anonymous publish/subscribe message-passing middleware
  - Communications are asynchronous
- Some modules will publish a set of topics while others subscribe to that topic
  - When new data is published, the subscribers can learn about the updates and can act on them
- Communication is implemented using a message-passing approach that forces developers to focus on clean interface logic
  - Described in the message interface definition language (IDL)
- ROS supports the recording and playback of messages
  - Messages can be recorded to a file and then played back to republish the data at any time
    - Allows for repeatability and facilitates regression testing

Core ROS Components #2

- Support for remote procedure calls via services
  - While asynchronous communications via pub/sub is great, sometimes you need lock-step synchronous behaviors
- Distributed parameter system
  - Tasks can share configuration information via a global key-value store
    - Provides a centralized point for changing configuration settings and the ability to change settings in distributed modules
- Robot-specific features like a geometry library, mapping and navigation functions, diagnostics and much more
- Extensive diagnostics capabilities

ROS Concepts

- ROS has three levels of concepts
  - Filesystem level
  - Computation level
  - Community level
- The filesystem level encompasses resources you'll likely encounter on disk
  - Packages
  - Metapackages
  - Package manifests
  - Repositories

Message (msg) types

Service (srv) types

ROS Concepts #2

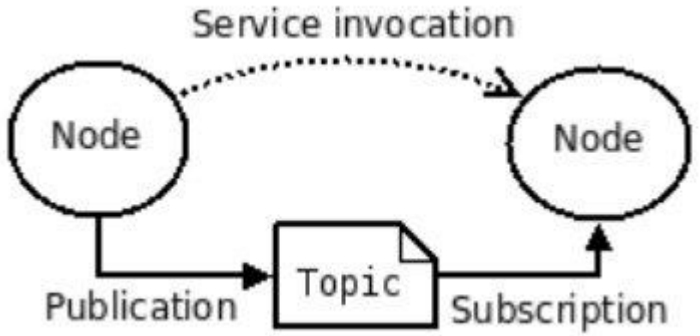
- The Computation Graph is the peer-to-peer network of ROS processes that are working together
- ROS computation graph level concepts include:
  - Nodes
  - Master
  - Parameter server
  - Messages
  - Topics
  - Services
  - Bags (places to store collected data)
- The ROS community-level concepts facilitate the exchange of software and knowledge between members of the community
  - Distributions
  - Repositories
  - The ROS Wiki
  - Bug ticket system
  - Mailing lists
  - ROS Answers (FAQ site)
  - Blog (information on updates including videos and photos)

Filesystem Specifics

- Packages are the primary unit of software in ROS (finest granularity)
  - Contains ROS runtime processes known as nodes, libraries, data sets, configuration files and anything else that's needed at this level
- Metapackages are a means to collect packages into related groups
- The package manifest (package.xml) provides the package name, version, description license, dependencies and other metadata related to the package
- Repositories are collections of packages that share a common version control system
  - Can be released as a unit using the bloom tool and may be mapped into rosbuild Stacks
- Message types describe the message data structures to be sent
- Service types define the request/response data structures for the service-level entity in ROS

Computation Graph Level

- Nodes are the processes that perform computation
  - Very fine granularity such as motor control, lidar interface, graphical view, etc.
- The Master is the clearing house for name registration and lookup to the rest of the graph
- Parameter server allows data to be stored, by key, in a central location and is typically part of the master
- Messages are the primary unit of communication in ROS and are data structures made up of primitive types (integers, floating point, booleans, etc.) and can be nested



Source: ros.org

Computation Graph Level #2

- Topics represent the messages that are routed via the pub/sub semantics
  - Node subscribe to topics while others publish topics
    - Supports one-many, many-to-many transport
- Services are the implementation of the RPC mechanism for synchronous communications in ROS
- Finally, bags are a format for record/playback of ROS message data and are the primary mechanism for storing sensor data

Naming Structure

- The communications graph and its components are represented in a global namespace that looks like a directory structure
  - / is the top level
- Resources are defined in their namespace and may define and share other resources
  - Resources can access anything in their namespace as well as those above their namespace
- Resources in different namespaces can be connected or integrated with code above both name spaces
- Typically code stays in its own namespace to preclude accidentally accessing objects of the same name in a different namespace
  - Each name is resolved locally as though each domain was a top-level domain
- Names can begin with ~, / or an alpha character (upper or lower)
  - Subsequent characters are alphanumeric, _ or /

Name Resolution

- There are four types of resource names in ROS
  - Base, relative name, global name and private names
    - Base name: base — Names with no namespace qualifier
    - Relative name: relative/name — Name relative to the local namespace
    - Global name: /global/name — Fully qualified names
    - Private name: ~private/name — Names that are not visible outside the namespace
- By default, all name resolution is relative to the local namespace
- Package resource names take the form of <packagename>/<msgtype>
  - E.g., std_msgs/String would be the String message type in the std_msgs package

Describing Robots in URDF

- The Unified Robot Description Format (URDF) is an XML-based way for representing a robot model
- The ROS URDF package contains XML specifications
  - All connections, mechanisms, subsystems, etc. must be described in URDF
    - Can get really tedious
- They have developed Xacro (XML Macros) as an XML-based macro language to simplify the definition of large robotic systems
  - Xarco helps reduce duplication of information in the file

Example: Building a Basic Chassis

- Two basic URDF components are used to define a simple robot chassis
- The **link** component describes a rigid body based on its physical properties
  - Dimensions, position in space, color, etc.
- Links are connected by **joint** components that describe the characteristics of the connection
  - E.g., Links connected, types of joint, degrees of freedom, axis of rotation, amount of friction, etc.
- The URDF description is a set of these link elements and their associated joint elements that connect the links together

A Simple Box in URDF

```
<?xml version='1.0'?>
<robotname="elc_robot">
  <!-- Base Link -->
  <link name="base_link">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
          <box size="0.5 0.5 0.25"/>
      </geometry>
    </visual>
  </link>
</robot>
```

A box that is .5m long, .5m wide and .25m tall
Centered at the origin of (0,0,0)
No rotation in the roll, pitch, or yaw (rpy)

Create the Package

- We need to create a package for this URDF to be placed

```
$ catkin_create_pkg elc_robot
Created file elc_robot/package.xml
Created file elc_robot/CMakeLists.txt
Successfully created files in
/home/mike/catkin_ws/src/elc_robot. Please adjust the values
in package.xml.
$ cd ~/catkin_ws
$ catkin_make
<lots of build output>
```

Create the urdf Directory and Populate it

- In the elc_robot directory, we create a urdf directory for the model XML

```
$ cd src/elc_robot
$ mkdir urdf
```

- Copy the URDF model into the urdf directory
- In order to run the model, we need a launch specification (also in XML) that can be passed to the **roslaunch** command
- We'll be using a simple visualizer called **`rviz`** to get started
- Create a launch directory and then create a **`elcrobot_rviz.launch`** as shown on the next page

```
$ mkdir launch
$ vi elcrobot_rviz.launch     ; use your favorite editor ☺
```

Create the Launch File

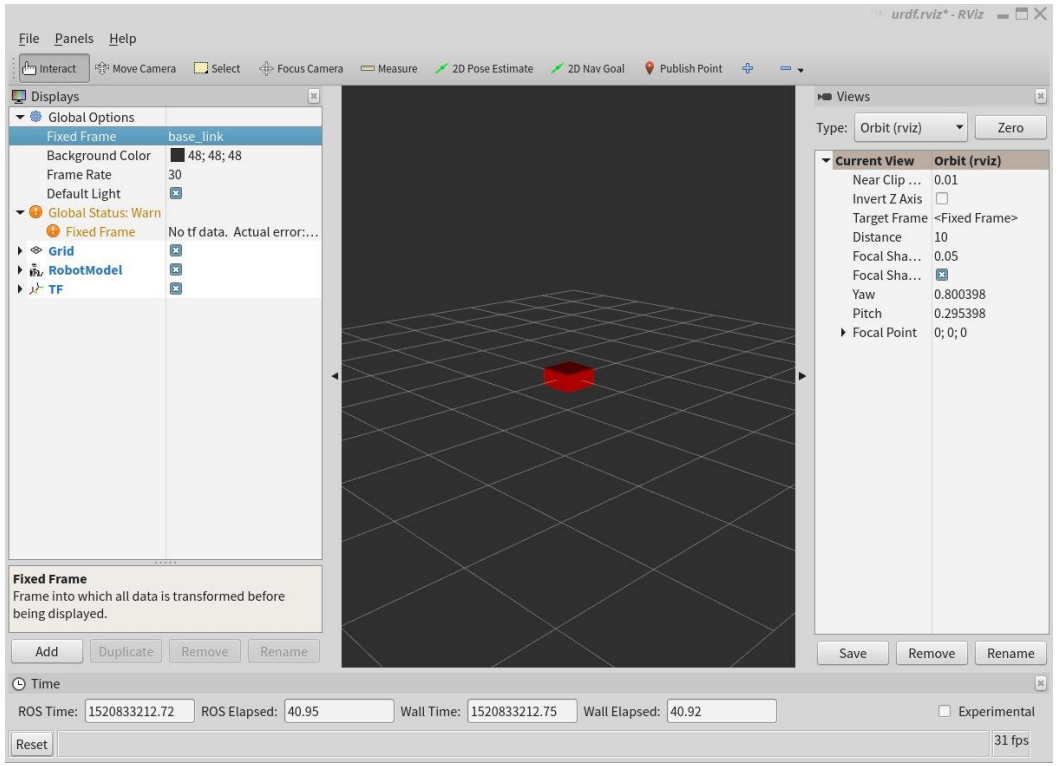
- Here is an example of a launch file:

```
<launch>
   <!-- values passed by command line input -->
   <arg name="model" />
   <arg name="gui" default="False" />

   <!-- set these parameters on Parameter Server -->
   <param name="robot_description" textfile="$(find elc_robot)/urdf/$(arg model)" />
   <param name="use_gui" value="$(arg gui)"/>

   <!-- Start 3 nodes: joint_state_publisher, robot_state_publisher and rviz -->
   <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />

   <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />

   <node name="rviz" pkg="rviz" type="rviz" args="-d $(find elc_robot)/urdf.rviz" required="true" />
   <!-- (required = "true") if rviz dies, entire roslaunch will be killed -->
</launch>
```

Launch the Model in all its Glory!

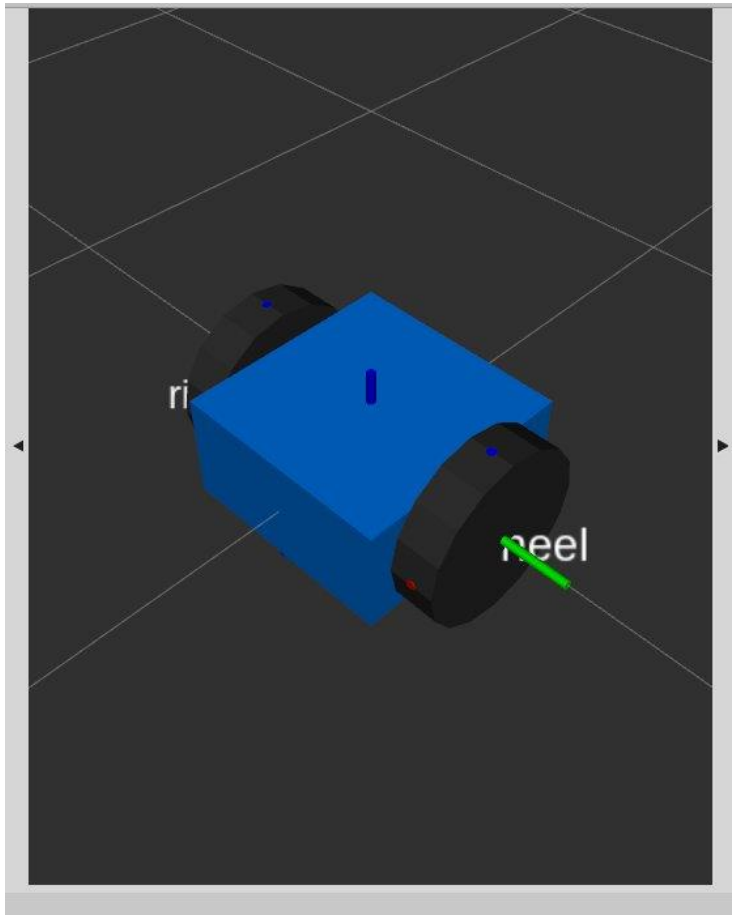
```
$ cd  ~/catkin_ws/src/elc_robot/
$ roslaunch elc_robot \
  elcrobot_rviz.launch \
  model:=elc_robot.urdf
```

- Wow, that's a lot of work for a box!
- But, it gets better!
  - Let's put some wheels on it and color it something other than red
- We'll need to describe the wheels, their radius, the joint connection to the base_link, their inertia, collision characteristics and mass



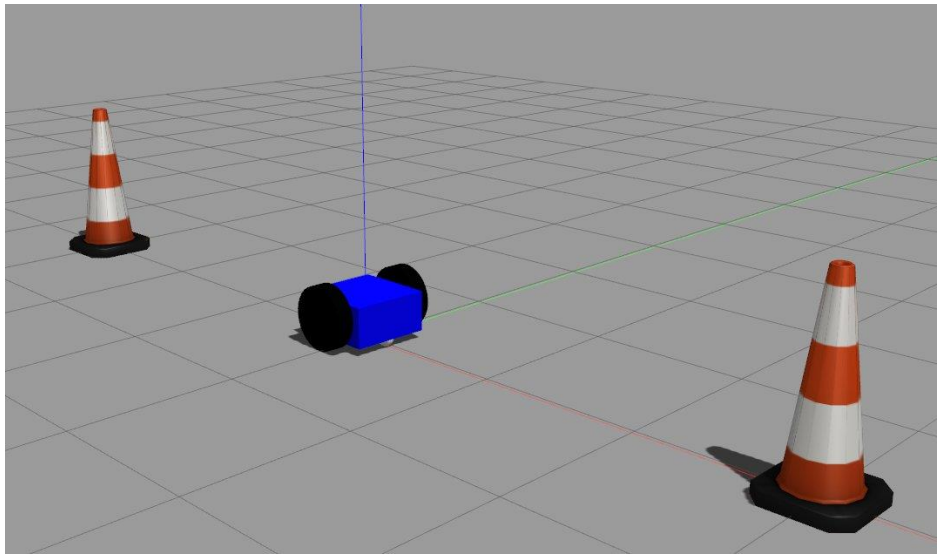
Box with Wheels!

- After making all of the necessary modifications, we have:
- Clearly, there is a lot of set up to define the robot and all of its connections
- But, once that's done, we can actually drive it around using **gazebo**



Gazebo

- ROS is compatible with a 3-D world simulator known as **gazebo**
- With gazebo, you can take the model you've built and place it into a simulated world so you can drive it around, manipulate gravity, etc.
- Gazebo is a separate install unless you install the "full_desktop" version of ROS initially

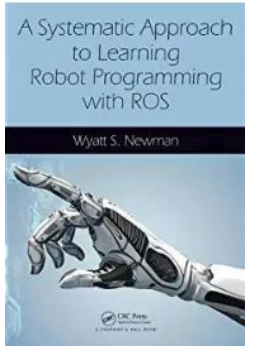


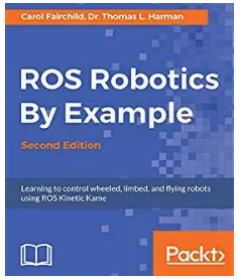
Example Pub/Sub

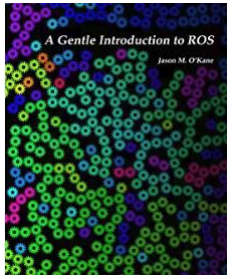
- The ROS wiki has a simple Pub/Sub example tutorial at:
  - http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29
- Walking through the code can be most enlightening because you get to see the definition of a message and the process for publishing/subscribing
- Clearly, there's a lot more to all of this
  - But, at least it's a start

Summary

- This has been a whirlwind tour of a clearly complex piece of code
- We've merely scratched the surface on this
- Defining the geometries of the robot can be daunting
  - It's a lot easier to build it in the real world!
- But, having described all of the interfaces and the message types and interactions you will have a much better understanding of your robot
- Fortunately, there is a large community around ROS
  - So, lots of folks to answer your questions
- And, many good reference books







Questions?


Embedded Linux
Conference
North America


OpenIoTSummit
North America